Fast Structural Join with a Location Function[⋆]

Nan Tang[†] Jeffrey Xu Yu[†] Kam-Fai Wong[†] Haifeng Jiang[‡]

[†] The Chinese University of Hong Kong, Hong Kong, China
[‡] IBM Almaden Research Center, 650 Harry Road, San Jose, USA
{ntang,yu,kfwong}@se.cuhk.edu.hk; jianghf@us.ibm.com

Abstract. A structural join evaluates structural relationship (parent-child or ancestor-descendant) between XML elements. It serves as an important computation unit in XML pattern matching, such as twig joins. There exists many work on efficient structural joins. In particular, indexes can expedite structural joins by skipping unmatchable elements. A typical use of indexes is to retrieve, for a given element, all its ancestor (or descendant) elements from an indexed set. However we observed two possible limitations with such index probes, namely *false hit* and *false locate*. A false hit means that an index probe touches unnecessary data besides real results; a false locate stands for a (wasted) probe that has zero answers. Obviously false hit and false locate can affect negatively the efficiency of structural joins. In this paper, we challenge ourselves to develop new structural join algorithm with *no false hit* and *no false locate*. We illustrate that *R-Tree* has the no false hit property (in contrast to B^+*-Tree*) and hence is a good candidate for our algorithm. For no false locate, we propose a new function called *Location* which tells the probing points that will result in matches. We design and implement the Location function using a space-efficient structure, and present our algorithm using *R-Tree* together with the Location function. Extensive experiments show the efficiency of our algorithm.

1 Introduction

Structural join is known as an important computation primitive in XML query processing. The Stack-Tree join algorithm proposed in [1] improved the traditional merge based algorithms with stack mechanism. Only one sequential scan is needed for two input ordered lists *A-List* and *D-List*. Index-based algorithms [2,3] improve the join performance. The essential idea is to use indexes on the participating element sets to directly (or near directly) find the matching elements and skip those without matches.

Despite of their success in performance improvement over merge-based algorithms, the current indexed-based algorithms are bound to two limitations, namely *false hit* and *false locate*. Given an element, find its ancestor (or descendant) elements from an indexed set, through an index probe. A *false hit* means

[⋆] This work is partially supported by the CUHK strategic grant (numbered 4410001) and the grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (CUHK418205).

that an index probe touches unnecessary data besides real results; a *false locate* stands for a (wasted) probe that has zero answers. *False hit* and *false locate* can affect negatively the efficiency of structural joins.

We summarize the contributions of this paper as follows:

1. We make a comparison between *R-Tree* and *B^+-Tree* on their support to structural searches. We conclude that structural search on *R-Tree* indexed data does not incur *false hit* while that on *B^+-Tree* has *false hit*.
2. We propose a `Location` function to accurately locate the matchable elements. It is built on top of a succinct and space-efficient bit-vector structure, namely *Locator*, which stores the distribution information of element nodes in a set and can be used to retrieve index probing points for no *false locate*.
3. We present a new index-based structural join algorithm, *R-Locator*, which combines *R-Tree* and *Locator* to achieve no *false hit* and no *false locate*.

Section 2 presents the motivation. Then Section 3 compares *R-Tree* over *B^+-Tree* on indexing region encode. We propose in Section 4 the space-efficient bit-vector index *Locator* and show how to use *Locator* for no *false locate*. Section 5 proposes a new structural join algorithm based on *R-Tree* and *Locator*. Section 6 analyzes the experiments. Finally Section 7 concludes this paper.

2 Motivation

Stack-Tree-Desc/Anc [1], is a milestone in structural join algorithm by maintaining an in-memory stack, with which we need only once scan of two input ordered lists. Many optimization comes from the observation that many unmatchable elements may attend the join operation depending on the characters of documents and queries. To efficiently skip these unmatchable elements, two approaches, *B^+-Tree* and *XR-Tree* based structural join algorithms, are proposed in [2,3] respectively. Consider the ancestor-descendant query $a//d$, Figure 1 shows the cases for different skip apporaches. Without indexes, the *Stack-tree* join algorithm will retrieve all the a elements and d elements, including $a_1 \dots a_{11}$, $d_1 \dots d_5$. The results are just (a_7, d_1) and (a_{11}, d_5), many nodes are *false hit*.

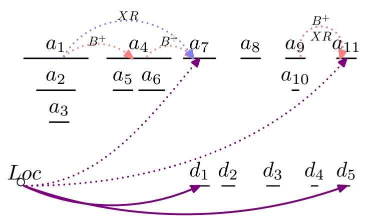

Fig. 1. Skip Elements Cases

Chien [2] utilized the property that if a tree element x is not the ancestor of z, then any descendant y of x cannot be the ancestor of z either. We use $S(e)$ and $E(e)$ to represent node e's *startpos* and *endpos* under region encode, respectively. With *B^+-Tree* index, when a_1 is not the ancestor of d_1, we could find the first element e satisfying $S(e) > E(a_1)$ and $S(e) < S(d_1)$, so to get a_4 as shown in

Figure 1, similarly we could get a_7, etc. With this skip technique, however, many useless elements are still *false located*, including $a_1, a_4, a_8, a_9, d_2, d_3, d_4$.

XR-Tree proposed in [3] could keep extra structural relationship so that, given an element e and an indexed set R, all e's ancestors (or descendants) in R can be identified efficiently. With *XR-Tree*, we could jump from a_1 to a_7 when we judge that a_1 is an unmatchable ancestor element and d_1 is stabbed by a_7. However, still many unmatchable elements are *false located* as $a_1, a_8, a_9, d_2, d_3, d_4$. *XR-Tree* outperforms B^+*-Tree* in the point that the less hit of a_4.

The reason of *false locate* of both B^+*-Tree* and *XR-Tree* is that they only grasp the local structural information instead of the global structural information, which drives the design of `Location` function. The goal of `Location` function is that, when we get a_1 which has no matches, we could retrieve d_1 and a_7; when we get d_2 which has no matches, we could retrieve d_5 and a_{11}.

3 *R-Tree* vs. B^+*-Tree* : No False Hit vs. False Hit

Encoding Selection: Consider the two most commonly used encodings: Region encoding and Dietz encoding. We use $S(e)$ and $E(e)$ to represent node e's *startpos* and *endpos* under region encoding, respectively. One special character of region encoding is that, for an element x, $(S(x), E(x))$ is a region, any descendant y of x, its region $(S(y), E(y))$ is covered by $(S(x), E(x))$. We adopts region encoding in this paper to show the advantage of this unique character.

3.1 R-tree for No False Hit

Figure 2 shows an XML data tree with region encoding. In general, one index structure is built for each tagname of element. For example, if we use B^+*-Tree*, we have one B^+*-Tree* for a and one for d. It is similar when using *R-Tree*.

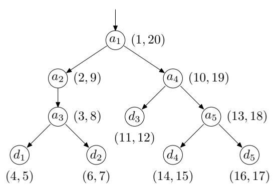

Fig. 2. An XML Document

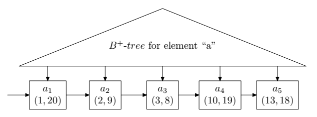

Fig. 3. An B$^+$-tree Index

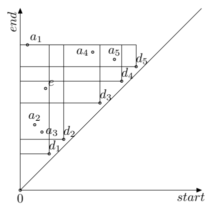

Fig. 4. R-tree Retrieve Area

Figure 3 shows a B^+*-Tree* index for a. If we want to retrieve the ancestors of d_3, we can search on it with *key* value less than $S(d_3) = 11$, and get a_1, a_2, a_3, a_4 while not the results a_1, a_4. This is called *false hit* and validation is needed. Region encode is a 2-dimensional data, using 1-dimensional index structure (B^+*-Tree*), we lost some information when dropping dimension.

As is well known, *R-Tree* is an excellent high-dimensional index structure, especially for 2-dimensional data. Figure 4 shows the areas to be retrieved under *R-Tree*. Review above example, when querying the ancestors of d_3, only the rectangle area including a_1 and a_4 is returned without redundant data. This is also the truth for any other element and other XPath axes. Therefore, *R-Tree* produces no *false hit* for XPath axes.

With above analysis, we may conclude that *R-Tree* outperforms B^+*-Tree* in scanning. The *false hit* of B^+*-Tree* retrieves all possible data including not only all the results but also redundant data; while the *R-Tree* with no *false hit* merely retrieve the results. To further optimize the *R-Tree* performance in structural join, we utilize the bulk-loading process and *R-Tree* packing [4] techniques to keep the *start* order in leaf page when we insert node in increasing *startpos*. This leads to a full storage utilization in the *R-Tree* leaves and consequently improves the query performance.

3.2 Stack Mechanism for R-tree

Assume the stack is represented as $\mathcal{T}$, the stack operations for B^+*-Tree* proceed as follows: first get a_1 and d_1, push a_1 into $\mathcal{T}$, and then get a_2, compare a_2 with $\mathcal{T}.top()$ to see whether $S(a_2) > E(\mathcal{T}.top())$ for popping stack. We get $false$ for this operation so next we compare $S(a_2)$ and $S(d_1)$, we *push* a_2 into $\mathcal{T}$ because $S(a_2) < S(d_1)$. The operations of ds are similar,for a new d, we compare it with $\mathcal{T}.top()$ iteratively to pop elements in $\mathcal{T}$ that are not the ancestors of d. The deficiency of B^+*-Tree* is the redundant comparison operation together with potential redundant *push* and *pop* stack operations coming from the *false hit*.

R-Tree also outperforms B^+*-Tree* in this aspect as follows. We get the first element of descendant d_1, then retrieve on *R-Tree* to get a_1, a_2 and a_3 with ascending order on *start* value. Then we could directly do the operation $\mathcal{T}.push(a_1)$, $\mathcal{T}.push(a_2)$, $\mathcal{T}.push(a_3)$. We insure the *start* order through bulk-loading process and *R-Tree* packing. a_1, a_2 and a_3 are insured to be the ancestors of d_1 because of the *no false hit* of *R-Tree*. Next we explain why a_1 must be a_2's ancestor and a_2 must be a_3's ancestor without comparison. See Figure 4, assume there is another node e which is d_1's ancestor, a_1's descendant and a_2's sibling node. Now d_1 have two ancestors a_2 and e that are sibling relationship, this contradict the character of tree structure. Thus this kind of node does e not exist, which insures the correctness of our put operation. We also save time on *pop* operation because we have no *false* elements in $\mathcal{T}$.

4 Locator for No False Locate

4.1 Locator : The Structure

Table 1 shows a group of assumptive encodings which is better than Figure 2 for explaining *Locator*, here N represents node name and C represents encoding. We build a *Locator* for each distinct tagname e, denoted as $\mathcal{L}_e$.

We maintain two *Locator*s for ancestor and descendant, separately. Figure 5 shows the *Locator* for element d, $\mathcal{L}_d$ and element a, $\mathcal{L}_a$. Each bit just represents the region length 1. Initially all bits are set to 0, if some element d_i intersects with

N	C	N	C	N	C	N	C	N	C	N	C
a_1	(1,6)	a_2	(2,5)	a_3	(3,4)	a_4	(7, 12)	a_5	(8,9)	a_6	(10,11)
a_7	(13,16)	a_8	(19, 20)	a_9	(23,26)	a_{10}	(24,25)	a_{11}	(29,32)	d_1	(14, 15)
d_2	(17,18)	d_3	(21,22)	d_4	(27,28)	d_5	(30, 31)				

Table 1. Encoded Elements

some bits of $\mathcal{L}_d$, the corresponding bits are set to 1. Even if each bit represents only length 1, the *Locator* is still a very space-efficient structure, for a large XML document, 100M for example, the number of elements is about 10^6 and the region range is about 2×10^6, about $200K$, which is not a problem for current memory capacity. Furthermore, we could use many mature techniques for compressing bit-vector structure.

L_a	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	1	0	0	1	1	1	1	0	0	1	1	1	1
L_d	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	1	0	0	1	1	0	0	0	0	1	1	0	1	1	0

Fig. 5. Locator for a and d

For instance, d_1's encoding is $(14, 15)$, so the 14th and 15th bits of $\mathcal{L}_d$ are set to 1. The construction of $\mathcal{L}_a$ is similar. Note that a bit may be set to 1 many times but only the first operation takes effect.

With $\mathcal{L}_d$, we could shrink the search space of *R-Tree*. Assume the query is $a//d$ for ancestor-descendant relationship, instead of using the tree height measure to shrink the query window, we could shrink the query window to $\{(29, 29), (32, 32)\}$ for d_5, here $(29, 29)$ is the coordinate of the lower left corner of rectangle query window and $(32, 32)$ is upper right corner. The query windows for other elements are similar. Sometimes when there are some region continuous ancestors such as a_1, a_4 and a_7, the shrink using tree height h may be better. We denote the region of the continuous 1 by r, so if $\frac{r}{2} \leq h$, we use the region code to shrink the query window, otherwise we use h.

4.2 Optimized Locator for No False Locate

Can we perform *no false locate*? The answer is positive. There seems to be two ways as shown in Figure 1, one way is to *locate* a_7 and a_{11}, the other way is to *locate* d_1 and d_5. We trace the second way. The basic *Locator* $\mathcal{L}_d$ cannot do this because it could only locate d_1, d_2, d_3, d_4, d_5 for *false locate*. Therefore we propose an optimization technique represented as: $\mathcal{L}'_d = \mathcal{L}_a \& \mathcal{L}_d$

The optimized $\mathcal{L}'_d$ is shown in Figure 6. With $\mathcal{L}'_d$, we could easily locate d_1 and d_5, which has *no false locate*. Then we could directly retrieve only a_7 and a_{11} with the *no false hit* property discussed above. This is a perfect structural join process.

L'_d	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0

Fig. 6. Optimized Locator for d

The '&' operation is safe, which means no results are lost. Note that this optimization technique could only be used on $\mathcal{L}_d$ while not on $\mathcal{L}_a$, which reflects the dissymmetrical relationship between a and d in ancestor-descendant structural query. We explain this phenomenon as follows. When some continuous 1 of $\mathcal{L}_d$ are set to 0 affected by $\mathcal{L}_a$, the descendants whose *start* values in these revalued bits must have no ancestors. Otherwise, it violates the containment relationship of the ancestor and descendant's region encoding: the region of ancestor's encoding always covers that of its descendants. For $\mathcal{L}_a$, in contrast, if some continuous 1 in $\mathcal{L}_a$ correspond to 0 in $\mathcal{L}_d$, it is very possible that some elements a whose *start* values fall in this area will have descendants in others regions. It also validates that the region of continuous 1 of the ancestor's *Locator* always covers that of its descendants. Therefore the opposite optimization $\mathcal{L}'_a = \mathcal{L}_d \& \mathcal{L}_a$ is not *safe*, which will lost some information to trace ancestors. This disproportional relationship between ancestor and descendant should be seriously considered in many cases about structural join using region encoding.

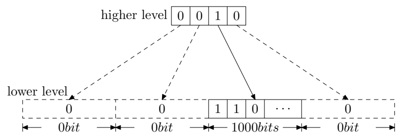

Fig. 7. Hierarchical Locator

An Analysis of Locator: Assume the page size is 4K (32768 bits), for a large XML document, say, 10^6 elements, we need only $\frac{2\times10^6}{32768} \approx 61$ pages, about $244K$, which could be easily loaded into main memory. If the XML document is huge, however, 10^9 elements for example, the size of *Locator* is $244M$. We could easily build a hierarchical *Locator* structure, we use 1 bit in the higher level to represent 1000 bits of its lower level; we set the higher bit 0 if all the lower 1000 bits are 0 but we store the lower 1000 bits as a virtual structure for saving storage space, otherwise we set it 1, which is shown in Figure 7. We preprocess the higher level to guide which part of the lower level should be read, and the higher level is just $244K$ that will be stayed in the main memory.

The cost of *Locator* is easy to analyze. Assume the number of element is $|X|$, the space and time complexities of using *Locator* are $O(|X|)$ for both $\mathcal{L}_a$ and $\mathcal{L}_d$ because at most one scan each is enough for performing all *location*. The I/O complexity analysis is straightforward as well. Each page of *Locator* is read once, hence we get the I/O complexity $O(\frac{|X|}{B})$, where B is the blocking factor.

5 Structural Joins using *R-Tree* and *Locator*

In this section, we propose a structural join algorithms *R-Locator*. When some elements which have no matches are met, we use *Locator* to perform the *Location* function and adopt *R-Tree* instead of B^+*-Tree* for retrieving elements.

Basic *Locator* Operation: The operation $\textsc{FindFirst}(L, pos, val, orien)$ is just a bit-vector operation. $\textsc{FindFirst}()$ is used to find the first *val* in L from a given position *pos* in the direction *orien*. We need scan the *Locator* only once, thus the upper limit of operating the *Locator* is the size of *Locator*.

R-Locator Structural Join Algorithm: *Locator* is a self-governed structure for `Location` function, which could be seamlessly combined with other index structures such as B^+-*Tree* and *R-Tree*. We use *Locator* to efficiently find the position where the descendant elements must have matchable ancestors and use B^+-*Tree* or *R-Tree* to retrieve the physical encodings. Here we select *R-Tree* for indexing because *R-Tree* could perform no *false hit*. *Locator* could also help to improve the performance of B^+-*Tree* in locating but it is not optimal. For instance, assuming we have three encoded ancestors $a_1(40, 45), a_2(50, 60), a_3(61, 70)$ and two encoded descendants $d_1(30, 35), d_2(63, 67)$; with $\mathcal{L}_d$, we know that d_2 must have matchable ancestors, and then we find corresponding continuous 1 in $\mathcal{L}_a$, which is $(50, 70)$. This is what *Locator* could do, with B^+-*Tree*, we will get a_2, a_3; while with *R-Tree*, we could get the result a_3. *R-Locator* algorithm is listed as:

Algorithm 1 *R-Locator* (A, D)

```
input: $A$: the ancestor set and $D$: the descendant set
output: Query results of $A//D$
 1: $a := First(A)$;   $d := First(D)$   $stack := \emptyset$;   $\mathcal{L}_d := \mathcal{L}_d \& \mathcal{L}_a$
 2: while $(a \neq End(A) \wedge d \neq End(D)) \vee \neg stack.empty())$ do
 3:   if $(a.start > stack \rightarrow top.end) \wedge (d.start > stack \rightarrow top.end)$ then
 4:     $stack \rightarrow pop()$
 5:   else if $(a.start < d.start)$ then
 6:     $stack \rightarrow push(a)$;   $a := a \rightarrow next$
 7:   else if $(\neg$stack.empty()$)$ then
 8:     output pairs$(a \in stack, d)$
 9:   else
10:     $pos :=$ FindFirst$(L_d,\ d.end + 1,\ 1,$ True$)$
11:     $d := Rtree_d \rightarrow find(pos, pos)$
12:     $pos :=$ FindFirst$(L_a,\ d.start - 1,\ 0,$ False$)$
13:     $pos := max(a.start + 1, pos + 1)$
14:     $a := Rtree_a \rightarrow find(pos, pos)$
15:   end if
16: end while
```

The algorithm keeps two cursors, a and d for current elements checked. The stack is adopted to hold the elements in A; and we also maintain the ancestor-descendant relationship in *stack*. The idea of this algorithm is also based on stack mechanism, but when elements having no matches are met, better than previous approaches to skip unmatchable elements and test the elements that may have matches, we adopt *Locator* to perform the `Location` function (line 10) and use *R-Tree* to find ancestors (line 14).

6 Performance Evaluation

6.1 Experiment Setup

Our test-bed is an experimental database system which includes a storage manager, a buffer pool manger, B^+-*Tree*, *R-Tree*, *XR-Tree* and *Locator*. All the algorithms were coded with Visual Studio .NET. All the experiments were conducted on a Pentium IV 2.80GHz PC with 1024M RAM and a 80G hard disk, running Windows XP. The page size used is 4K and we use the file system as the storage. All the experimental results presented below were obtained with a fixed buffer pool size: 100 pages, just large enough to cache the hot elements.

We use synthetic data for all our experiments in order to control the structural and consequently join characteristics of the XML documents. We adopt two DTDs based on *DBLP* DTD and *Department* DTD which is similar to that in [3]. We generated two 20M raw data for each DTD using the IBM XML data generator with default parameters.

6.2 No False Hit vs. False Hit

This group of experiments was conducted to study the performance of *R-Tree* vs. B^+*-Tree* for *false hit*. We first use optimized $\mathcal{L}'_d$ to exactly locate descendants, then we search the possible bit range on $\mathcal{L}_a$ for possible ancestors, and then we use *R-Tree* or B^+*-Tree* for retrieving the ancestors. We use the stack here to store the nested ancestors, so the number of ancestors retrieved by *R-Tree* is the exact number of results. The queries selected are listed in Table 8, we choose these queries to thoroughly test all the nested and distributive cases.

	Query	Document
Q_1	*employee//email*	*Department*
Q_2	*employee//name*	*Department*
Q_3	*inproceedings//title*	*DBLP*
Q_4	*book//homepage*	*DBLP*
Q_5	*article//homepage*	*DBLP*
Q_6	*article//editor*	*DBLP*

Fig. 8. Sample Queries

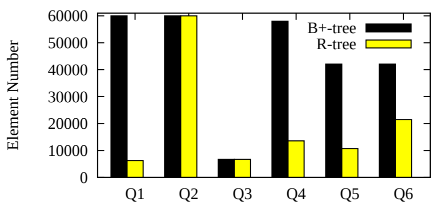

Fig. 9. *R-Tree* vs. B^+*-Tree*

Figure 9 shows the number of ancestors retrieved by *R-Tree* and B^+*-Tree*. If the ancestor elements whose ranges are covered by $\mathcal{L}_a$ are all matches of corresponding d got from $\mathcal{L}_d$, both *R-Tree* and B^+*-Tree* could get *no false hit*. Therefore the number of ancestors retrieved by *R-Tree* and B^+*-Tree* is the same, as shown in Figure 9 of the cases Q_2 and Q_3. In most cases, otherwise, some ancestor elements have continuous encodings but some are not matches of corresponding d got from $\mathcal{L}_d$, in which case *R-Tree* will still perform *no false hit* while B^+*-Tree* will get many useless ancestor elements because of *false hit*. Therefore B^+*-Tree* will retrieve much more ancestor elements than that of *R-Tree*. In Figure 9, the queries Q_1, Q_4, Q_5, Q_6 show this case.

6.3 Varying Selectivity on Elements

The objective of this set of experiments is to study the capabilities of various algorithms to skip elements. Among all experiments, we show the results for the case where the join selectivity on the ancestor set and the descendant set varied together, starting from 90% down to 1%.

Varying Selectivity on Ancestors: The first group of experiments is to vary join selectivity on ancestors. During this experiments, we kept the percentage of descendants that match at least one ancestor high (99%) and varied the selectivity on ancestors, i.e., the percentage of ancestors that have descendants. Figure 10(a) and 10(b) display the elapsed time for various algorithms. As can be observed from the figures, elapsed time with all algorithms decrease with the decrease of selectivity on ancestors. All approaches improve the performance by skipping unmatchable elements. Furthermore, *R-Locator* has the best overall performance. We also notice that *XR-Tree* is better than B^+*-Tree* in the case, this is because that *XR-Tree* is structured by maintaining stab lists in the ancestor, thus *XR-Tree* performs less *false locate* than that of B^+*-Tree*. One interesting to note that even only few elements can be skipped, despite of the possible overhead of index probing, all approaches perform no worse than the basic *Stack-tree* join algorithm.

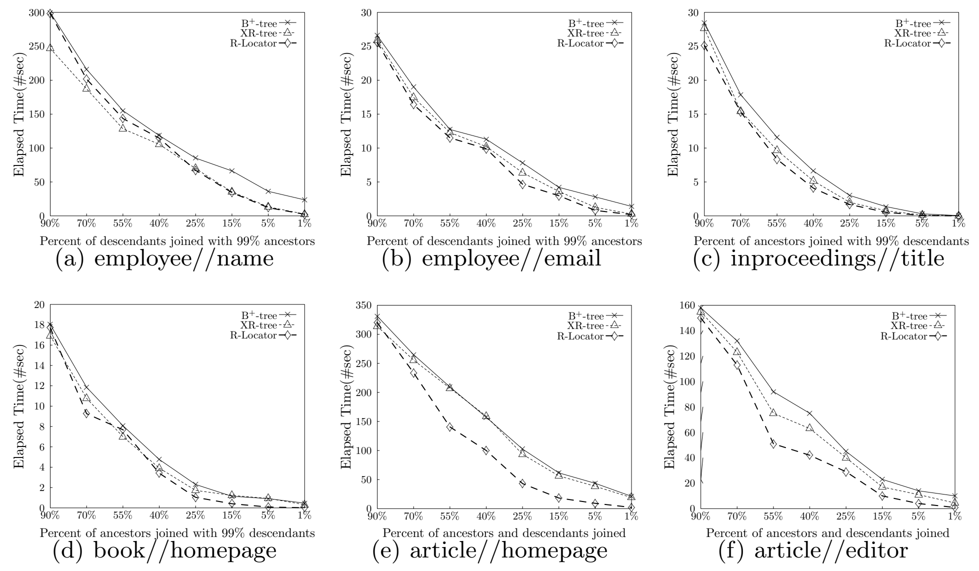

Fig. 10. Elapsed time (in second) for different join selectivity. (a) (b): 99% of descendants join with varying proportion of ancestors; (c) (d): 99% of ancestors join with varying proportion of descendants; (e) (f): varying proportion of ancestors and descendants are joined.

Varying Selectivity on Descendants: The second group of experiments is to test the performance when varying join selectivity on descendants. We keep the join selectivity on ancestors high (99%) and vary the join selectivity on descendants. Figure 10(c) and 10(d) show the overall performance of the algorithms tested. As observed from the figures, *R-Locator* perform the best performance. The *XR-Tree* is better than B^{+}*-Tree* in that it keeps more information than B^{+}*-Tree*, therefore has the less chance for *false locate*. While one potential higher overhead of *XR-Tree* is caused by two additional fields (ps, pe) in key entries, hence more index pages.
Varying Selectivity on both Ancestors and Descendants: In the last set of experiments, we vary the join selectivity on both ancestors and descendants. The results for these experiments are shown in Figure 10(e) and 10(f). As shown in the figures, *R-Locator* still performs the best overall performance. This can be explained as follows: as we vary the selectivity on both ancestors and descendants, the case of interleaving elements of unmatchable elements on ancestors and descendants increases, which increases the number of inevitable *false locate* on both B^{+}*-Tree* and *XR-Tree*. However, the distribution of elements has no effect on *Locator* for *no false locate*.

The diversity of the algorithms is best illustrated by this group of experiments, where there is potential to perform *Location* function. *Locator* could perform the *Location* function with low overhead, therefore it provides the best performance among all. *XR-Tree*, on the other hand, keeps more information than B^{+}*-Tree*, so it has more chance to achieve the *Location* function, thus it performs the second and B^{+}*-Tree* performs the worst overall performance.

7 Conclusion

In this paper, we propose a new `Location` function for fast structural join. Previous approaches are all failed in performing this function, the reason is that they use index to test whether the elements retrieved have matches by skipping unmatchable elements. The design of our space-efficient structure *Locator* is to efficiently perform `Location` function. The extensive performance evaluation show the significance of our proposed algorithm over previous approaches.

References

1. S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proceedings of ICDE '02*, 2002.
2. S.-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo. Efficient structural joins on indexed XML documents. In *Proceedings of VLDB '02*, 2002.
3. H. Jiang, H. Lu, W. Wang, and B. C. Ooi. XR-Tree: Indexing XML data for efficient stuctural join. In *Proceedings of ICDE '03*, 2003.
4. I. Kamel and C. Faloutsos. Updates for structure indexes. In *Proceedings of the 2nd Int'l Conference on Information and Knowledge Management(CIKM)*, 1993.